

CRAY T3E Programming with Coherent Memory Streams

Version 1. 2

Cray Research Inc.

OVERVIEW

Maintaining data coherence with the memory streams on the CRAY T3E is not automatic. If used improperly, data incoherence and processor or even system hangs can result. As a fail-safe mechanism the use of streams is turned off by default on the CRAY T3E. This is a complete solution for many applications.

However, executing CRAY T3E applications with the streams option turned off can impose a performance penalty for some applications. Not all applications can make substantial use of the streams option. Candidate applications are those in which important kernel portions make short stride references to local memory. For such codes, some programming adjustments can render them fully stream-safe, and thus regain top performance.

Library changes in place by January of 1997 should allow the streams option to be on by default for detectably stream-safe applications. At that time many single processor codes and all PVM or MPI codes will automatically be able to take advantage of the streams option.

CODE ADJUSTMENTS

A simplistic description of the adjustment needed to make a code stream-safe is to prevent simultaneous cached and uncached references to closely spaced memory locations. With the streams option turned on, small-strided, regular references (loads or stores) by a CRAY T3E processor to its local memory can cause prefetching to the stream buffers for subsequent loading to the ev5 processor cache. Each processing element (PE) can also asynchronously reference any PE's memory using E-register uncached load and store commands ("get"s and "put"s). The stream-based, cached memory references and the asynchronous uncached memory references might be operating simultaneously. The object of code adjustments should be to eliminate any such simultaneous memory references to similar addresses. By similar we mean two locations in one PE's memory that are within 24 (64 bit) words of each other. Specifically, to be stream-safe an uncached reference should be to an address that is less than or at least 192 bytes greater than that of simultaneous, cached, stream-based references to the local memory. (See Figure 1.) Remember, this unsafe addressing zone only applies when the streams option is on and in use and another (or much less likely, the same) processor makes a simultaneous uncached reference to the local memory.

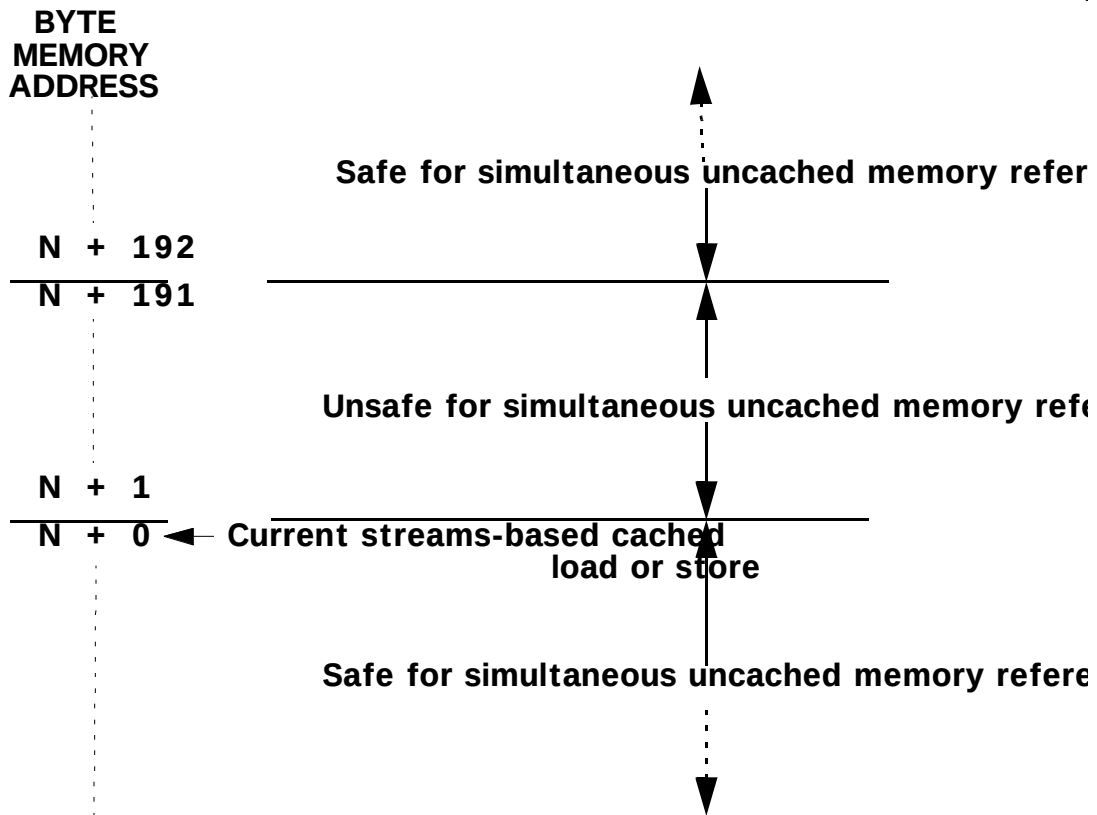


Figure 1. Local memory zones relevant to stream-safe references

There are two situations in which a remote PE might be addressing the local PE's user space memory at the same time as local cached memory references (see Figure 2):

- * Multi PE applications involving explicit data exchanges via "shmem", PVM, or MPI library calls (or lower level direct accessing of E-registers). PVM and MPI message passing codes using release 1.1.0.1 or later of the Message Passing Toolkit (MPT) will be guaranteed stream-safe. Until then,

codes employing MPT should be considered candidates for the remediations described below.

* Single and multi PE applications employing streams during asynchronous I/O, that is, I/O which uses system calls to read, write, or listio.

Note that here the “remote PE” may in fact be OS or I/O device services.

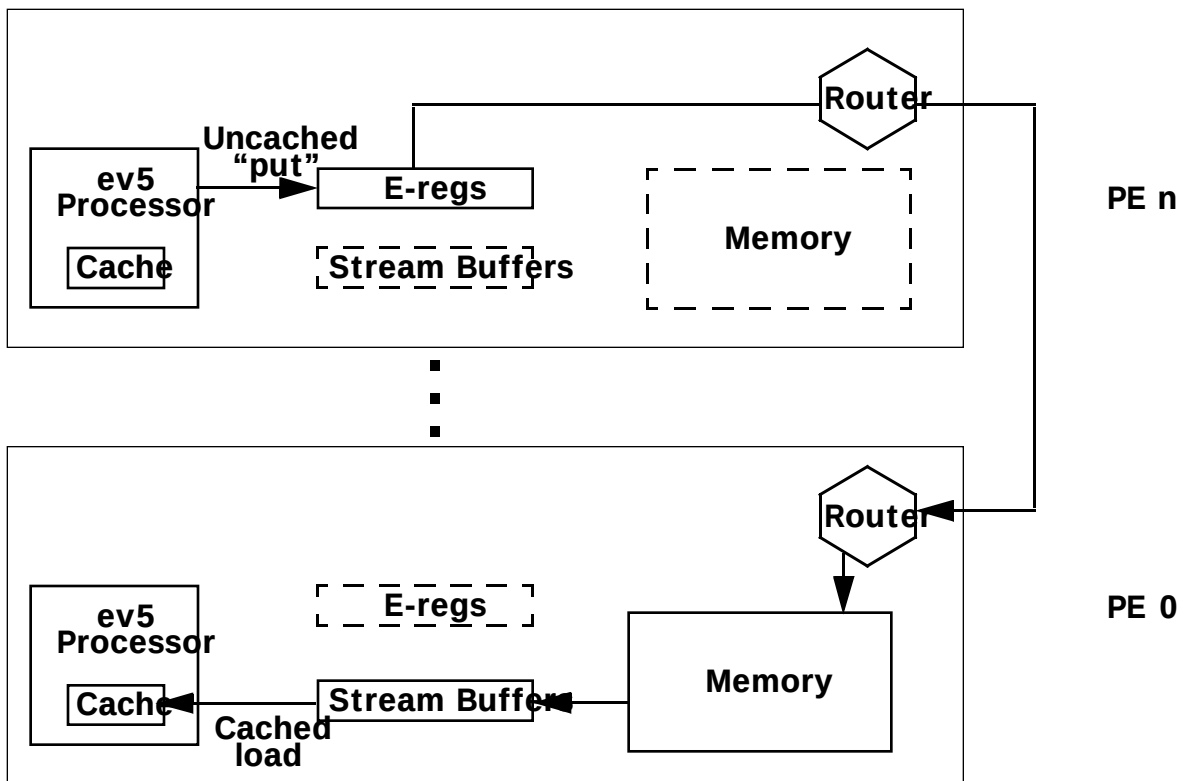


Figure 2. Schematic example of multiple processor data paths leading to possibly simultaneous cached and uncached references to common memory. PE 0 is involved in local stream-based loads while PE n stores to PE 0's memory using uncached stores. PE n alternatively might be an IO port or OS PE, processing asynchronous input or output for PE 0.

In addition, there are three situations in which a single PE, as part of a single or multi PE application, might provide simultaneous referencing (see Figure 3):

- * Applications involving local memory access using “shmem” library calls (or lower level direct accessing of E-registers). The local referencing, for example, could be part of a global addressing scheme involving memory locally and on other processors.

- * Applications which use cached loads or stores to local memory and then closely follow this with “shmem” PUTs of the data from that local memory. The PUTs of this local data, probably to locations in another processor’s memory, invoke uncached GETs of the local memory.

- * Applications which use the CACHE_BYPASS compiler directive or pragma, which invokes E-register uncached local memory access.

The above five cases apply to codes written in C, C++, Fortran-90, or Cray Assembler for MPP (CAM) for the CRAY T3E. The generic solutions are common across all languages. However, the following will address high level language cases and examples only.

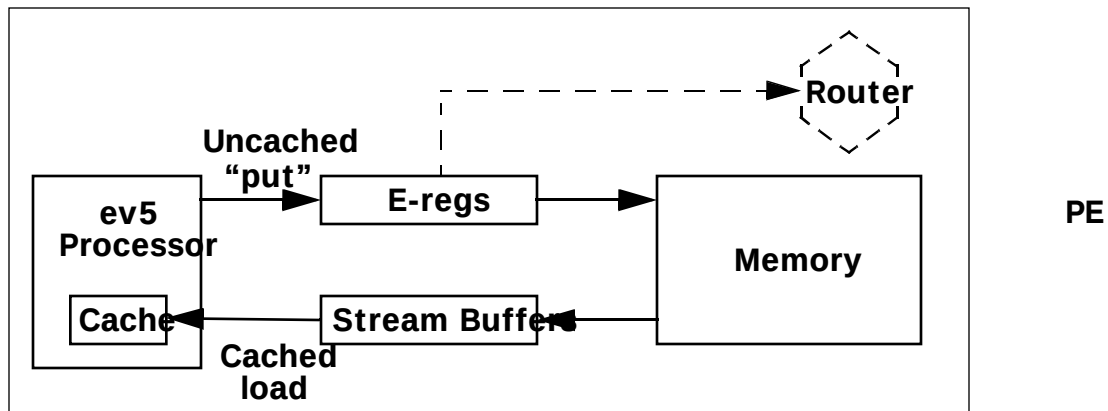


Figure 3 . Schematic illustrating possible data paths on a single PE which is involved in simultaneous cached and uncached references to common memory. In this example, the PE has stream-based loads overlapping with puts to local (and possibly global) memory.

Note that the HPF model on the CRAY T3E involves implicit data exchanges which are not guaranteed to be stream-safe. Thus, users of HPF on the CRAY T3E are encouraged to leave the streams option off.

The incoherence of the CRAY T3E memory streams can occur only in the specific cases described above. An application programmer can remedy all such by either (1) not using streams or (2) separating cached and uncached references in space or time.

1. Not Using Streams.

As pointed out earlier this is the rather obvious solution for many codes but also a quite undesirable solution for others. However, their use is not an all or nothing situation. Users have simple, flexible control over the use of streams. One can write

```
CALL SET_D_STREAM( 1 )      --- Fortran
```

```
set_d_stream( 1 );          --- C or C++
```

at any point in a code that use of data streams is desired and include the statement

```
CALL SET_D_STREAM( 0 )      --- Fortran
```

```
set_d_stream( 0 );          --- C or C++
```

to turn the streams option off (see man set_d_stream(3)). Turning streams on at an application's onset permits the streams to be active for the entire application. More relevantly, a programmer can safely choose to activate the streams hardware logic only in sections of code in which its performance boost is key and simultaneous cached and uncached memory references can not occur. Be aware that once a stream pattern has been recognized by the CRAY T3E support hardware, data will continue to flow through the streams buffers until the pattern is broken. Turning streams off once they have been turned on merely prevents the support logic from recognizing and establishing new streams. The 2.0.3 programming environment release will include an additional function

```
CALL QUIET_D_STREAM()      --- Fortran
```

```
quiet_d_stream();          --- C or C++
```

which turns the streams option off and “parks” the support logic. This function breaks the data flow from the streams buffers. Calls to quiet_d_stream require about 10 microseconds but fully guarantee subsequent memory coherency. An example usage in Fortran is given by

```
! Code with cached loads and stores here
```

```
IPREV = GET_D_STREAM()
```

```
IF (IPREV .GT. 0) CALL QUIET_D_STREAM ! park streams
```

```
CALL BARRIER()           ! synchronize
```

```
! Code with interleaved computation and communication here
```

```
CALL BARRIER()      ! synchronize  
CALL SET_D_STREAM(IPREV) ! restore data streams
```

2a. Separating References in Space

Since coherence of streams is in doubt only when close-by addresses are simultaneously referenced, judicious spacing can eliminate potential problems. For example, if array A is used in a local computation while a related array, B, is a communications buffer for shmem or asynchronous MPI data movement, padding between A and B guarantees stream coherency. Thus, in Fortran

```
INTEGER A(1000), B(1000), PAD(24)  
COMMON /HLDR/ A, PAD, B
```

or in C or C++

```
#define PAD 24  
  
long arr[1000 + PAD + 1000];  
  
long *A = arr;  
  
long *B = arr + 1000 + PAD;
```

the arrays A and B are spaced safely, even if A is accessed with local cached references and B is accessed with uncached references. This is similar to the optimization technique used for spacing of concurrently addressed arrays to minimize cache line overwriting. Such padding suffices for both purposes. Note also that placing the local computation array, A, at higher starting address than B, e.g.

```
COMMON /HLDR/ B, A
```

provides the same coherency guarantee according to the code adjustment guidelines above. A simple pad of 192 bytes might be the easier method to remember, but either technique is equally valid.

The separation of references in space is also a sufficient solution for guaranteeing stream-safe code involving concurrent, asynchronous I/O. The common practice of double buffering for asynchronous I/O is rendered stream-safe by padding the start of each buffer with 24 64-bit words.

Note that distributed I/O to be released on the T3E as DISTIO uses listio and will not be stream-safe. DISTIO allows I/O to be initiated to or from a remote PE and thus is a form of asynchronous I/O. Global I/O, another, yet to be released I/O feature on the T3E, will be guaranteed stream-safe. Global I/O is a distributed buffering mechanism that makes use of shared, global file pointers.

2b. Separating References in Time

Although you need to allow for simultaneous references of memory for asynchronous messaging and I/O, turning off the streams option or separating references in space may be impractical in some situations. Stream-based loads or stores and uncached memory references commonly use portions of the same array(s). For example, in spatially partitioned problems ghost cells of computed variables might be updated by other PEs during continued computation of adjacent cells on the host/owner PE. Alternatively, one PE may be moving data to a common array while other PEs move their contributions into adjacent or interleaved locations in the same array.

The best code adjustment in such cases is to synchronize the activities to prevent overlapped references in time. Although many strategies of speeding up or slowing down of activities may be tried, the most straightforward is the use of barriers (see `man barrier(3C)`, `shmem_barrier(3)`, or `pvmbarrier(3)`). A barrier on a CRAY T3E is a function which returns control only after all members in its PE group (generally all PEs) have called the function and all inter PE communications are completed. Barriers are very fast on the CRAY T3E and provide an easy to use time synchronization technique.

The following code segment includes a striped movement of data into the SORT array and movement of remote data into the same array. The call to the BARRIER function separates these two activities and thereby guarantees stream-safe references to the SORT array.

```
INTEGER MYPE, NPES, ASIZE, M_TO
REAL A(ASIZE), SORT(*)
!DIR$ SYMMETRIC SORT
DO I = 1, ASIZE
    SORT((I-1)*NPES+MYPE+1) = A(I)
ENDDO
CALL BARRIER() ! <--- separates local and remote moves
M_TO = MOD((MYPE+1),NPES)
CALL SHMEM_IPUT(SORT(MYPE+1),
*             SORT(MYPE+1), NPES, NPES, ASIZE, M_TO)
```

For similar circumstances in which only a subset group of PEs participate in such overlapping activity, the shmem and pvm libraries provide barrier functions that synchronize only that group. Thus, the similar code would be written

```
INTEGER MYPE, NPES, ASIZE, M_TO, pSYNC(*)
INTEGER MYGRPSTART, LOGSTRIDE, NUMINGRP
REAL A(ASIZE), SORT(*)
LOGICAL MEMBER
!DIR$ SYMMETRIC SORT, pSYNC
IF (MEMBER) THEN
    DO I = 1, ASIZE
        SORT((I-1)*NPES+MYPE+1) = A(I)
```

```

        ENDDO    ! barrier separates local and remote moves

        CALL SHMEM_BARRIER(MYGRPSTART, LOGSTRIDE,
*
            NUMINGRP, pSYNC)

        M_TO = MOD((MYPE+1),NPES)

        CALL SHMEM_IPUT(SORT(MYPE+1),
*
            SORT(MYPE+1), NPES, NPES, ASIZE, M_TO)

    ELSE

        :

    ENDIF

```

A similar solution is afforded by a pair of barrier function calls when control branching allows only some PEs to perform remote data movements while other PEs make stream-safe memory references. For example,

```

INTEGER MYPE, NPES, ASIZE, M_TO

REAL A(ASIZE), SORT(*)

!DIR$ SYMMETRIC SORT

IF (MOD(MYPE,2) .EQ. 0) THEN

    DO I = 1, ASIZE

        SORT((I-1)*NPES+MYPE+1) = A(I)

    ENDDO

    CALL BARRIER() ! <--- signify local moves complete

ELSE

    CALL BARRIER() ! <--- possibly wait for remote moves

    M_TO = MOD((MYPE+1),NPES)

    CALL SHMEM_IPUT(SORT(MYPE+1), A, NPES, 1,
*
        ASIZE, M_TO)

```

ENDIF

An alternative method of effecting time separation of memory references is to create memory “lock” words. As each PE attempts to make local or remote memory references to a shared memory region, it checks the value of the lock word to determine the status of that block of memory. If no other PE “owns” that memory, the lock word will be in the null state and the querying PE can swap its ownership value into the lock word. If the lock word indicates current memory use, the querying PE can spin wait on the lock word or proceed to any other useful work. The `shmem_swap` function on the CRAY T3E (see `man shmem_swap(3)`) can be used to create very efficient multi processor lock words. One could rewrite the above data movement example code to use lock words instead of barriers as shown below. Here the purpose is to share usage of the global array, `SORT`, by having a single lock word, `LKSORT`, hold the ownership semaphore for the array. Each PE must establish ownership for the array, whether referencing the local or a remote `SORT` array. Note that `LKSORT` must be initialized, in this case to -1, by all PEs before beginning this segment of code. Note also that `LKSORT` itself is rendered fully stream-safe by placing it after a 24 word buffer in a common block.

```
INTEGER MYPE, NPES, ASIZE, M_TO, IRTN

INTEGER LKSORT    !Synchroniz. initialize to -1 previously

REAL A(ASIZE), SORT(*)

INTEGER SHMEM_SWAP, LPAD(24)

COMMON /LLOCKS/ LPAD, LKSORT !Assure lock word stream-safe

!DIR$ SYMMETRIC SORT

IRTN = 0

DO WHILE (IRTN .NE. -1) ! Own my lock word ?
```

```

        IRTN = SHMEM_SWAP(LKSORT, MYPE, MYPE)

ENDDO

DO I = 1, ASIZE

    SORT((I-1)*NPES+MYPE+1) = A(I)

ENDDO

IRTN = SHMEM_SWAP(LKSORT, -1, MYPE) ! Free my lock word

M_TO = MOD((MYPE+1),NPES)

IRTN = 0

DO WHILE (IRTN .NE. -1) ! Own M_TO's lock word ?

    IRTN = SHMEM_SWAP(LKSORT, MYPE, M_TO)

ENDDO

CALL SHMEM_IPUT(SORT(MYPE+1),

*           SORT(MYPE+1), NPES, NPES, ASIZE, M_TO)

IRTN = SHMEM_SWAP(LKSORT, -1, M_TO) ! Free M_TO's lock

```

The lock word technique is a viable alternative to barriers for creating temporal separation. It does have potential advantages over the barrier construct in some situations. The lock technique will have less cost than the barrier technique when a small number of PEs contend for each lock. This is accomplished when relatively fine-grained portions of a global array are protected by each lock. On the other hand, if many or all PEs must obtain the lock at about the same time, this high degree of lock contention will result in poor performance. In such cases use of a barrier would have less overhead.

The lock technique is more verbose and for that reason more prone to error. However, in complex codes with multiple logic paths, memory locks might be more straightforward than placement of judicious barriers.

In this case, use of locks is easier and, therefore perhaps, safer.

For some applications a global array into which the uncached memory references are made might border on memory space where local computation occurs concurrently. For such cases padding the start of the global array (with 24 64-bit words as suggested above) is required. Although the barrier or lock technique effectively separates PEs in time from referencing a global array, it might not separate such references from those of the local processor to adjacent memory .

A final alternative approach is provided by the function

```
CALL WAIT_D_STREAM()          --- Fortran  
wait_d_stream();              --- C or C++
```

also included in the 2.0.3 programming environment release. Calls to this routine cause a wait of approximately 1.3 microseconds until any outstanding incoherent stream buffer loads are completed. The `wait_d_stream()` function does not turn the streams option off but does provide the necessary time separation. This function can be used quite effectively immediately before uncached references which use “shmem” library calls. It guarantees coherency with preceding cached data loads or stores. Used together with a call to a barrier function such as the Fortran `CALL BARRIER()` or C `barrier()`, the `wait_d_stream()` function provides the time separation needed to prevent stream buffer incoherency for multi PE instances. Used alone, `wait_d_stream` ensures coherency for single PE code segments with potential simultaneous cached and uncached references.

A code segment application in C of `wait_d_stream()` is given by

```
for (i=0;i<32; i++)  
    a[i] = b[i];
```

```
wait_d_stream(); /* wait for stream prefetches in "a" */  
shmem_put((targ, a, 32, remote_pe);
```

SUMMARY

There are several situations on the CRAY T3E which allow applications to encounter the lack of automatic coherency in the streams option. By and large, encountering these situations will be quite rare. However, since the outcome is non-deterministic, some thought must be given to correct use of streams. The above describes the three methods available: restrict the use of streams, separate memory references in space, or separate memory references in time. There are trade-offs in the use of the three techniques, and only one may be viable with respect to desired performance in a given code.

A systematic approach might be taken to evaluating the alternatives. First, determine if selective or complete deactivation of the streams option is acceptable for each instance of potential streams incoherence. (This is obviously the simplest technique and presumably should be our first choice.) If leaving the streams option off is unacceptable from a performance viewpoint, then the programmer needs to examine the coding constructs outlined in the sections on spatial and temporal memory references separation. Padding or relocation in memory of referenced arrays and scalars will be the obvious next choice where the number and type of referenced structures is small and under the control of the application programmer. The third group of techniques involves the use of barrier and/or lock words. This is the most natural solution when the memory reference patterns are fairly complex or indeterminate. It might also afford the smallest possible change to the code that guarantees stream-safe memory references. None of these techniques are novel or complicated. The application of any or all, therefore, is generally quite straightforward.